

11 Using JUnit with jGRASP

jGRASP includes an easy to use plug-in for the **JUnit** testing framework. JUnit provides automated support for unit testing of Java source code, and its utility has made it a *de facto* standard among software professionals. JUnit is particularly popular among *test-first* and *test-driven development* (TDD) advocates. To successfully use JUnit with jGRASP requires knowledge of both jGRASP and JUnit. This tutorial assumes you have some experience using jGRASP but are new to JUnit. While the primary focus of the tutorial is on the operational features in jGRASP that pertain to JUnit, the examples are intended to provide a brief introduction to JUnit. If you are not familiar with the basic features of jGRASP (e.g., compiling, running, and debugging), you are encouraged to read the tutorial *Getting Started with jGRASP 2.0*. A web search on “junit” will turn up a wealth of information on it.

Objectives – When you have completed this tutorial, you should be able to do the following: (1) configure the JUnit plug-in for use with the version of JUnit that you have installed; (2) open a source file in jGRASP and create a corresponding test file; (3) create test cases (test methods) in the test file for each of the methods in the source file; (4) run the test file for a corresponding source file; (5) run all test files for a project; (6) view the results of running the test files in the JUnit window.

The details of these objectives are captured in the hyperlinked topics listed below.

11.1 Preliminaries

11.2 Creating, Compiling, and Running a JUnit Test File

11.3 Triangle Example

11.4 Notes on Testing with JUnit

11.4.1 Using jGRASP Interactions versus JUnit for Testing

11.4.2 Test Methods in JUnit

11.5 Other JUnit Resources

11.1 Preliminaries

For this tutorial, you need to have installed the Java JDK, jGRASP, and JUnit. If you have installed each of these, then skip to **4. JUnit Configuration in jGRASP**. Otherwise, follow the steps below as appropriate.

1. **Java JDK** – If you have already installed the Java JDK, go to the next step. Otherwise, install the Java JDK, which is available from Oracle at <http://www.oracle.com/technetwork/java/javase/downloads/index.html>. Be sure to download the full JDK rather than the JRE.
2. **jGRASP** – If you have already installed **jGRASP 2.0.1 or later** (which includes a plug-in for JUnit), go to the next step. Otherwise, download and install the latest version of jGRASP (<http://www.jgrasp.org/>).
3. **JUnit** – If you have already installed JUnit, go to the next step. Otherwise, download the ZIP file version (e.g., junit-4.12) of the current release of JUnit (<http://www.junit.org/>). Download the following JARs and put them in a folder with *junit* in the name (e.g., junit-4.12): [junit.jar](#), [hamcrest-core.jar](#)

The *junit* folder should be moved/dragged to the *program files* folder on Windows or an appropriate location such as */usr/bin* on other systems.

4. **JUnit Configuration in jGRASP** – After starting up jGRASP, on the main menu, select **Tools > JUnit > Configure**. The JUnit Tool Settings dialog should open as shown in Figure 1. If jGRASP was able to find JUnit in a standard location, the **JUnit Home** field will already be set to the folder containing the JUnit executables (e.g., C:\Program Files\junit-4.12). If **JUnit Home** is blank or incorrectly set, click the **Browse** button and set it to the folder containing the junit JAR file and hamcrest-core JAR file, then click OK. You should now be ready to use JUnit with jGRASP.

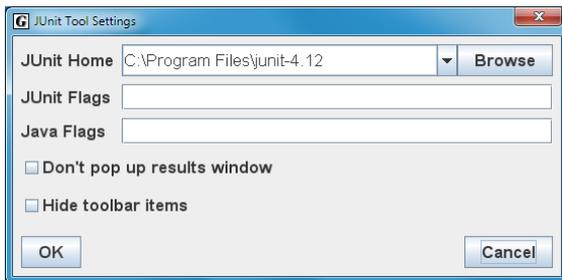


Figure. 1 Configuring the JUnit Plug-in

11.2 Creating, Compiling, and Running a JUnit Test File

The procedure for using JUnit to test a java source file is summarized below:

1. In the Browse tab, navigate to the folder containing the source file. If the source file is not in a jGRASP project, create a project using the Project menu (**Project > New**), and then add the source file to project. Also, add any other source files that are part of the same project. After this, you should see the open project and source file(s) in the Open Projects section of the Browse tab.
2. Open the source file in jGRASP by double-clicking on the file name.
3. On the top toolbar, click the Create Test File  button. Alternatively, on the top menu you can click **Tools > JUnit > Create Test File**. [Note that if an existing file in the folder has the same name as the test file to be created, jGRASP will ask if you want to use the existing file as the test file rather than create a new test file.] After the test file has been created, this menu option and button becomes **Edit Test File**. Note that the test file naming convention is the source file name with “Test” appended to it (e.g., for source file Triangle.java, the test file should be TriangleTest.java).
4. Test cases are added to the test file in form of test methods. Note that the defaultTest() method provided by jGRASP asserts that 0 equals 1 (i.e., the *expected* value is 0 and the *actual* value is 1) so that the test will fail. This test method should be replaced by your own relevant test methods. The **Triangle Example** below describes the details of creating, compiling and running test methods.
5. The test file is compiled and run by using the JUnit buttons on the toolbar:  to compile the test file and  to compile (if needed) and run the test file. Alternatively, you can use top menu: **Tools > JUnit > Compile Tests** and **Tools > JUnit > Run Tests**. If you want to compile or run all test files for a project, you can click the JUnit button  and/or  on the Open Projects toolbar.
6. When a test method fails (assuming the test method is correct), set a breakpoint on the statement in the test method where the expected result is determined – typically this will be a statement that calls a method in the source file being tested. Now run the test file in debug mode . After the program stops at the breakpoint, step-in  at the method call and then single step in the source file looking for the statement that has resulted in the incorrect return value. Correct the error (a.k.a., fault or defect) and run the test file again, hopefully with better results.

“Marking” a Test File – If a test file is not created using jGRASP as indicated in the steps above (i.e., the file was created outside of jGRASP, or it was created as a regular source file), it must be marked as a test file so that jGRASP will know to compile and run it as a test file. This is done by adding the test file to the jGRASP project. jGRASP automatically distinguishes between source and test files as they are added to the project, and it places them in appropriate categories. **If a test file is listed in the Source Files category for jGRASP project, it is not yet marked as a test file.** To mark a file in the Source File category as a test file, right-click on the file and select “Mark as Test” (the file should move to the Test Files category). When both categories are present, you can also simply drag files from one category to another.

11.3 Triangle Example

Let’s consider an object-oriented version of the classic triangle program. Given three sides (a, b, and c), we must determine: (1) if the sides can form a triangle; i.e., all sides are greater than zero, and no single side is greater than the sum of the other two; and (2) if we have a triangle, classify it as equilateral, scalene, or isosceles.

The source code for this example is included with jGRASP. If you already have a copy of the jGRASP examples in a user folder, navigate to the folder now: ...\\jgrasp_examples\\Tutorials\\JUnitExamples\\TriangleExample folder.

Otherwise, use the jGRASP Browse tab to navigate to the folder where you want to copy the examples, then click **Tools > Copy Example Files**. The dialog should open in the same folder as the Browse tab (otherwise, navigate to the folder where you want to save the examples) and click Choose. Find the jGRASP examples in the Browse tab.

1. Navigate to the TriangleExample folder by opening Tutorials, then JUnitExamples, and finally the TriangleExample folder.
2. **Open the project TriangleExample.gpj** file by double-clicking on the file. Now you should see the project in the Open Projects window in lower part of the Browse tab. Unfold the Source Files and Test Files to show Triangle.java and TriangleTest.java in their respective categories as shown in Figure 2.

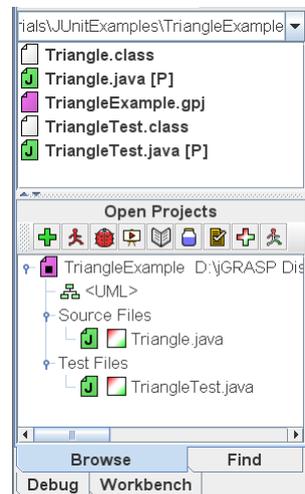


Figure. 2 Open Projects in Browse Tab

3. **Open Triangle.java** by double-clicking on the file in Source Files category of the Open Projects window. This class includes three double fields a , b , and c for the three sides, a constructor, and a classify method. The constructor takes three parameters of type double representing the sides and then sets the sides and checks to make sure the sides can represent a triangle. Figure 3 shows the constructor and classify method of the Triangle.java opened in jGRASP. If you are not familiar with throwing and catching Java exceptions, ignore the two *if* statements in the constructor, and instead, concentrate on the classify method in the Triangle class.

```

public Triangle(double aIn, double bIn, double cIn)
{
    a = aIn;
    b = bIn;
    c = cIn;
    if (a <= 0 || b <= 0 || c <= 0) {
        throw new IllegalArgumentException("Sides: " +
            + " " + c
            + " -- each must be greater than zero.");
    }
    if ((a >= b + c) || (b >= a + c) || (c >= a + b))
    {
        throw new IllegalArgumentException("Sides: "
            + a + " " + b + " " + c
            + " -- not a triangle.");
    }
}

/**
 * Classifies a triangle based on the lengths of the
 *
 * @return the triangle classification.
 */
public String classify() {
    String result;
    if ((a == b) && (b == c)) {
        result = "equilateral";
    }
    else if ((a != b) && (a != c) && (b != c)) {
        result = "scalene";
    }
    else {
        result = "isosceles";
    }
    return result;
}

```

Figure 3. Triangle.java

4. **Open TriangleTest.java** by double-clicking on the file in the Test Files category of the Open Projects window of the Browse tab. TriangleTest.java contains import statements for JUnit, an empty setup method annotated with `@Before`, and 10 test methods, each annotated with `@Test` in the method header. Note that the annotations can be on separate lines if desired (`@Test` is shown both ways in the example). The first five test methods are checking for exceptions to be thrown correctly, and the last five are checking correct classification (equilateral, isosceles, and scalene). If you have not worked with Java exceptions, concentrate on the last five test methods. Figure 4 shows three of the 10 test methods in TriangleTest.java. These use one of JUnit's `Assert.assertEquals` methods where the first argument is an optional String that will be printed if the assert is false. The second argument is the *expected value*, and the third argument is the *actual value*, which is commonly the result of a call to a method in the class being tested (e.g., calling the `classify` method in the Triangle class).
5. **Compile and/or Run the Test File** by clicking the  and/or  on the toolbar. The Compile JUnit Tests button  compiles the test file and the

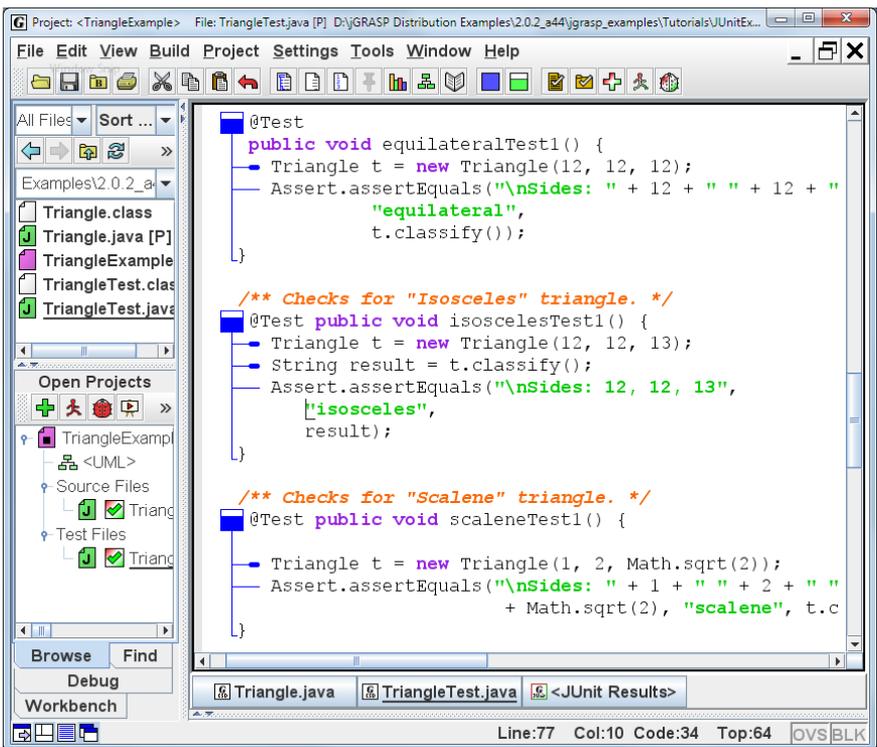


Figure 4. TriangleTest.java

Compile and Run JUnit Tests button  compiles (if needed) and runs the test file. If you want to compile and/or run all test files for a project, click the JUnit buttons  and/or  on the Open Projects toolbar. When you run a single test file or all test files in a project, the results are reported in the Run I/O tab as well as in a separate JUnit window. Figure 5 shows the results in the Run I/O tab and Figure 6 shows the JUnit window.

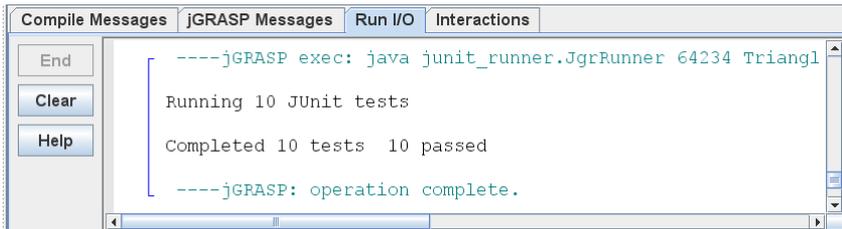


Figure 5. JUnit results in the Run I/O tab

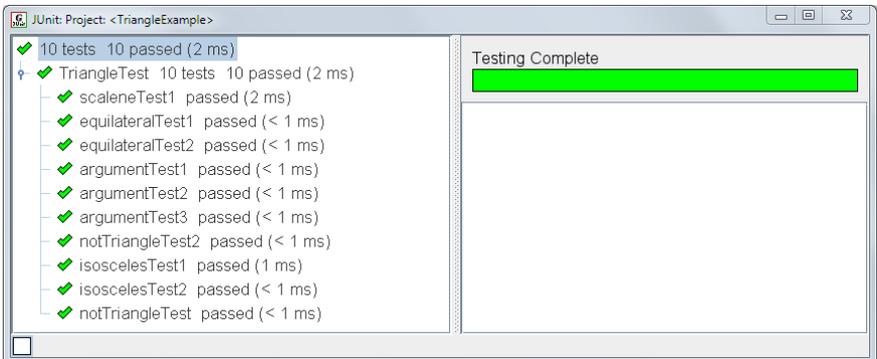


Figure 6. Test results in the JUnit window with test method unfolded

If all test methods in test file pass, a checkmark is displayed on the JUnit status in front of the test file and the associated source file in the Open Projects window as shown in Figure 7. If either file is edited after passing all tests, the checkmarks disappear indicating that the test file needs to be run again. Now let's introduce a simple spelling error in the classify method in Triangle.java by changing "isosceles" to "isoscele" on line 53 (press Ctrl-L for line numbers). Since Triangle.java has been edited, it must be compiled  before TriangleTest.java is run (if you forget, jGRASP will remind you).

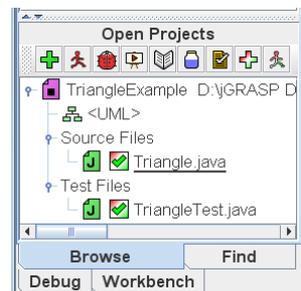


Figure 7. JUnit Status for classes showing all test methods passed

Now let's run  the test file again to see if our test methods are good enough to catch this error. The results are shown in Figures 8, 9, and 10.

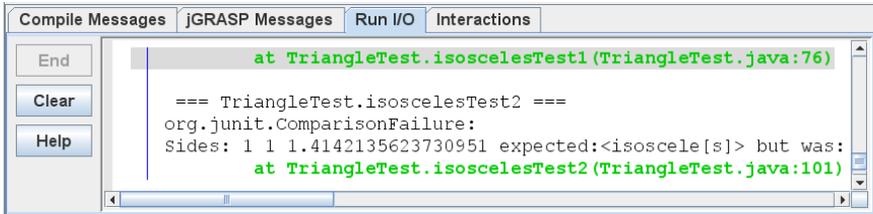


Figure 8. JUnit results in the Run I/O tab showing failures

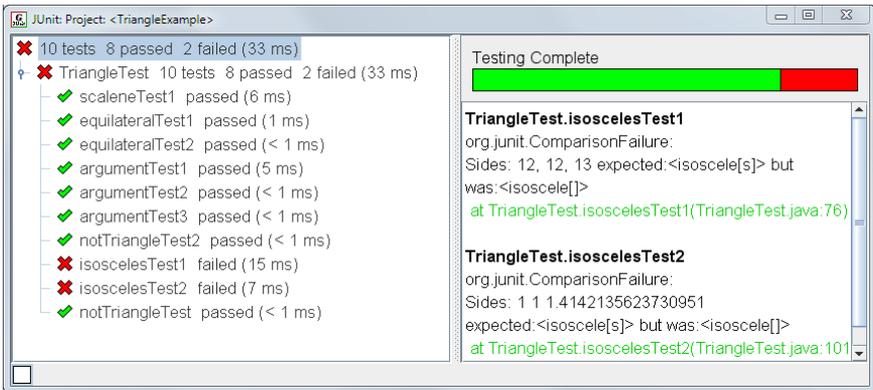


Figure 9. Test results in the JUnit window showing failures

When one or more failures occur, the first one is highlighted in the Run I/O window (see Figure 8) and the test file is scrolled to the assert statement that caused the failure. In this example, the message gives a clear indication of what went wrong: “Sides: 12, 12, 13 expected: <isoscele[s]> but was: <isoscele[]>”. It even indicates the two strings differed by an s. You can use the links in the Run I/O window or the JUnit window to quickly find any of the assertions that failed in the source code. Assuming the test method is correct, we need to find the error in the source file.

To do this, set a breakpoint on the line in the test file where the method is called that yields the actual result. In the case of first failure, this would be the call to `t.classify()` on line 75 in `TriangleTest.java`. Now run the test file in debug mode . After the program stops at the breakpoint, step-in  on the

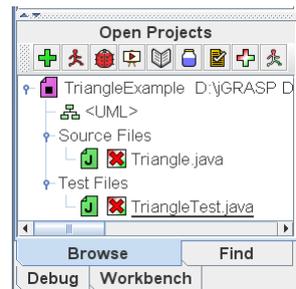


Figure 10. JUnit Status for classes showing one or more test methods failed

statement in which `t.classify()` is called and then single-step ↓, looking for the statement that should set the variable `result` to **"isosceles"**. Correct the error by changing **"isoscele"** back to **"isosceles"** on line 53. Compile + the source file, then run 🏃 the JUnit test file again and make sure all test methods pass. Usually, errors are more subtle than this one that we created, but the technique for finding them is the same: (1) set a breakpoint in the test file at the statement where the source method is called; (2) run the test file in debug mode 🐞. After the program stops at the breakpoint, step-in ↵ to the source method; (3) single-step ↓, looking for the statement that is responsible for the failure.

11.4 Notes on Testing with JUnit

Here are a few terms related to testing.

Unit Testing: Testing one unit or component at a time (e.g., testing a class and its methods).

Failure: An undesired (incorrect) result produced by the software.

Fault (or Defect): The underlying cause of the failure (a “bug” or “error” in your code).

The purpose of testing is to identify failures so that the underlying faults (or defects) can be removed.

Debugging is the process of removing a fault. (Note that debugging occurs after a failure has revealed the existence of a fault.)

11.4.1 Using jGRASP Interactions versus JUnit for Testing

jGRASP Interactions can be used to informally test methods in your class. For example, you can create a new `Triangle t` and then invoke the `classify` method on it to make sure the correct result is returned as shown in Figure 11.

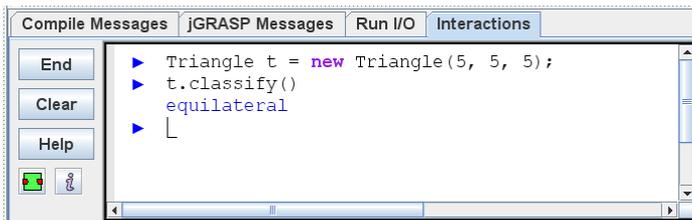


Figure 11. Using Interactions to test methods informally

Informal testing in Interactions is useful for small programs as well as for spot checking larger programs. However, as your programs become larger, it will become too tedious to use Interactions for testing and re-testing classes and

methods as you make changes to your source files. The JUnit Framework allows us to essentially formalize the approach of creating an instance, calling a method on it, and checking that the return value was correct. Using JUnit, you can build a test file and add test methods as you construct the methods in a source class. You can then re-run the test file whenever changes and/or additions are made to the class during development.

11.4.2 Test Methods in JUnit

In JUnit, the informal test above can be written as follows:

```
/** Checks for "Equilateral" triangle. */
@Test public void equilateralTest1() {
    Triangle t = new Triangle(5, 5, 5);
    Assert.assertEquals("\nSides: " + 5 + " " + 5 + " " + 5,
        "equilateral",
        t.classify());
}
```

The `@Test` tag marks the method as a test method and the JUnit `Assert.assertEquals` method is used to assert that the expected value of `"equilateral"` equals the return value of `t.classify()`, which is the actual value produced by the program. If a JUnit assert statement *fails*, (i.e., is not what was asserted), an `AssertionError` is thrown, and we see the results in the Run I/O tab and the JUnit results window. For the example above, our error message `"\nSides: " + 5 + " " + 5 + " " + 5` will be printed as part of the failure information provided by JUnit. Below, several JUnit assert statements are described with respect to the type of the items being compared.

When comparing integer values or objects:

```
Assert.assertEquals(expected, actual);
Assert.assertEquals(error msg, expected, actual);
```

When comparing float or double values:

```
Assert.assertEquals(expected, actual, delta);
Assert.assertEquals(error msg, expected, actual,
    delta);
```

Delta is the number of decimal points that you want to compare. For example, `0.000001` compares two doubles to six decimal places (10^{-6}), which means if the floating point values are within `0.000001` of each other, they are considered to be equal. Due to rounding, floating point values may not be exactly equal so you must provide a delta value. See the `TemperatureConverter` folder for an example of comparing doubles.

When comparing arrays of integer values or objects:

```
Assert.assertEquals(expected, actual);  
Assert.assertEquals(error msg, expected,  
actual);
```

When comparing arrays of float or double values:

```
Assert.assertEquals(expected, actual, delta);  
Assert.assertEquals(error msg, expected, actual,  
delta);
```

Asserting True or False:

```
Assert.assertTrue(boolean expression);  
Assert.assertTrue(error msg, boolean expression);  
Assert.assertFalse(boolean expression);  
Assert.assertFalse(error msg, boolean expression);
```

When comparing two String values for a partial match:

Suppose you want to know if the String `str1` contains **"scalene"** somewhere within it. The following will be true if it does.

```
Assert.assertTrue(str1.contains("scalene"));
```

11.5 Other JUnit Resources

For details on all assert methods see the Assert JUnit API:

<http://junit.sourceforge.net/javadoc/org/junit/Assert.html>

For information on JUnit in general see:

<http://junit.org/>